Naval Research Laboratory
Stennis Space Center, MS 39529-5004

NRL/MR/7441--96-8032

Object-Oriented Database Exploitation Within the GGIS Data Warehouse

MIYI J. CHUNG
KEVIN B. SHAW

Mapping, Charting, and Geodesy Branch
Marine Geosciences Division

MARIA A. COBB

Planning Systems Incorporated
Slidell, LA

DAVID K. ARCTUR
JOHN F. ALEXANDER

University of Florida
Gainesville, FL

January 17, 1997

REPORT DOCUMENTATION PAGE

Form Approved
OBM No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 17, 1997	Final

4. TITLE AND SUBTITLE

Object-Oriented Database Exploitation Within the GGIS Data Warehouse

5. FUNDING NUMBERS

Job Order No. 5745137A7
Program Element No. 0603207N
Project No.
Task No. R-1987
Accession No.

6. AUTHOR(S)

Miyi J. Chung, Kevin B. Shaw, Maria A. Cobb*, David K. Arctur†, and John F. Alexander†

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Marine Geosciences Division
Stennis Space Center, MS 39529-5004

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/7441--96-8032

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Chief of Naval Research
Code 824
800 North Quincy Street
Arlington, VA 22217-5050

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

*Planning Systems Incorporated, 115 Christian Lane, Slidell, LA 70458
†University of Florida, Gainesville, FL

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report presents an introduction to initial efforts in converting relational vector mapping data (Vector Product Format), produced and distributed by the National Imagery and Mapping Agency, into object-oriented mapping data. From a single Vector Product Format implementation, this work continued to include four more Vector Product Format databases into object-oriented framework. Discussion includes actual functionalities that were added, as well as results of having multiple Vector Product Format databases in object-oriented framework. With the possibility of growing mapping databases, a search for an object-oriented database management system was considered and evaluated.

14. SUBJECT TERMS

Tactical oceanography, dynamical oceanography, physical oceanography, electronic/electrical engineering

15. NUMBER OF PAGES

25

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

I. Introduction

This report documents the initial findings and accomplishments to date of the **Object-Oriented Database Exploitation within the GGIS Data Warehouse** project, sponsored by the Defense Mapping Agency (DMA). The primary goal of this project is to investigate, through research and prototyping efforts, the potential impact of object-oriented (OO) technology on DMA's Global Geospatial Information and Services (GGIS) modernization program. Specifically, the tasks for the project include:

- providing all findings that would aid DMA in the letting of the major production contract for the GGIS data warehouse
- producing an object-oriented prototype capable of importing multiple Vector Product Format (VPF) databases, performing updates on the data, and exporting the updated data to both relational and object-oriented formats
- connecting the prototype to commercial object-oriented database management systems (ODBMS's) to provide persistent storage and query/management capabilities

Background material concerning both object-oriented and relational databases is provided in Section II of this document. The material is intended to familiarize the reader with the data model, terminology, advantages, and disadvantages of both approaches. Section II also contains a description of the initial project set-up, including software and hardware utilized.

In Section III, project accomplishments to date are described, including prototype development status and preliminary research investigations conducted.

In carrying out the tasks mentioned earlier, key findings concerning the feasibility, desirability and practicality of using either OO, relational or hybrid technology for the foundation of the GGIS data warehouse are documented in detail in Section IV.

Section V provides a description of the design work performed in support of the object-oriented VPF (OVPF) prototype, upon which a major part of this work is based. This design includes a class hierarchy and description of the relationship among major classes.

Section VI follows with a statement of work remaining for FY95, while Section VII addresses possible future needs that this project could provide a basis for solving. Section VIII provides a summary of progress thus far, along with a prognosis for the year's remaining work.

Section IX furnishes our initial recommendations, based on the current status of the project, regarding the focus and direction of GGIS.

II. Background

Relational Databases

Currently, DMA's VPF specification is based on the relational database approach. Relational database models utilize relations, or tables, of data for modeling real-world entities. Each row (tuple) in a table represents a collection of values which together describe a single instance or entity of the relationship. Each column represents an attribute that occurs for every instance of the relationship. A tuple is the basic unit of information in relational databases. Operations such as insertion and deletion are performed at the tuple level. Each tuple can be accessed through the use of a **primary key**. A primary key is an attribute or collection of attributes, the value of which is guaranteed to be unique for given tables.

To model any non-trivial enterprise with a relational model, many tables must be used. Each table contains the information needed for all instances of a particular relationship; however, inter-relationships among tables often exist that must be captured in some manner. These inter-relationships are modeled through the use of **foreign keys**. A foreign key for a relation is an attribute or collection of attributes that acts as a primary key for another relation. Thus, foreign keys allow information from one relation to be combined with related information from other relations (i.e., foreign keys are the mechanism used to link actual files together).

The relational approach is a sound one since it is based on formal, mathematical concepts, which provide for a well-defined and understood model. Additionally, the relational algebra associated with the data model supplies a complete set of operations for manipulating relations for query purposes.

Many of the disadvantages of the relational approach are related to the fact that often information for a single entity may by necessity be scattered throughout multiple tables. For example, in VPF, information for each area feature is contained in separate relations, e.g., face, ring and edge tables. Figure 1 gives an example based on the VPF specification of how such information can be stored.

Although all of this information may be retrieved through the use of foreign keys as explained above, this technique has several drawbacks. One of the drawbacks is simply the extra *storage space* and *programming effort* required to maintain the foreign keys. Essentially, each key value for a relation must be stored and maintained not only in that relation, but in every single relation that wishes to reference the data in the original relation. Another concern is that of referential integrity. *Referential integrity* between relations refers to the concept that any tuple that is referenced from a relation must exist. This is particularly a problem when one considers updates. For example, if a tuple in a relation that is referenced from other relations is deleted, then that reference must be deleted in every relation in which it occurs. This is also a concern upon modifications to key fields (either primary or foreign), as well as upon insertion of new tuples. For almost all cases,

these integrity constraints must be maintained by the database application programmer.

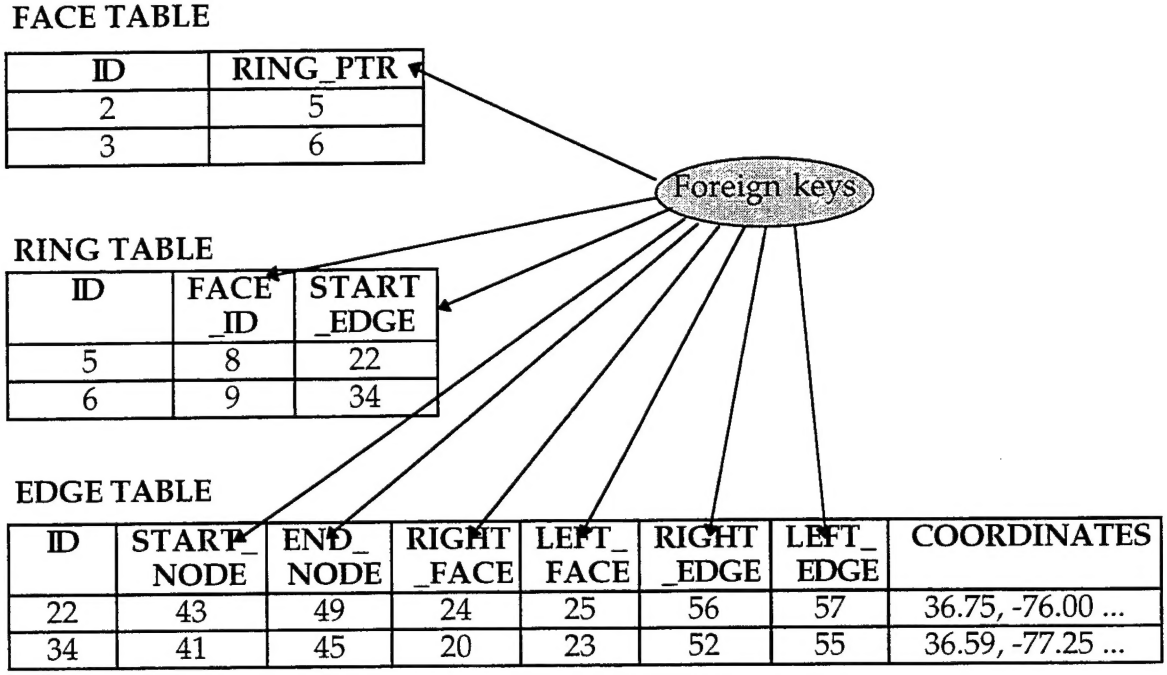

Figure 1. Relational table and foreign key example

The process of combining related tuples from different relations is known as a **join** operation. This operation is considered to be one of the most expensive, time-wise, operations that can be performed on relations; however, its use is necessary due to the fact, as mentioned earlier, that it is very often not possible to represent all of the information for a single entity in one table. Although efficient algorithms for performing joins have been developed, these still require at minimum a complete pass to be made over each of the tables involved.

One final drawback of the relational approach is its limitation with respect to modeling arbitrarily *deep data hierarchies*. For example, a state can be thought of as being composed of counties, which in turn are composed of sections, etc. In the relational approach, the only way available for modeling this composition hierarchy is through the use of foreign keys. This approach is not only non-intuitive, but it is associated with the problems described above.

Object-Oriented Databases

Object-oriented databases were originated to provide a solution to some of the inadequacies of the relational model. In object-oriented databases, each real-world entity is modeled as an **object**. Each object may in turn be composed of other objects, which have pointers to other objects, in a "containment hierarchy." These pointers may be for attribute data (analogous to columns in the relational approach), or direct links to other complex

objects (analogous to foreign keys in relational). This approach is more intuitive than the relational approach, in that each data component in an OO database corresponds directly to the feature being modeled, and contains all information relevant to that feature in a single, coherent structure. This feature alone eliminates the costly table joins necessary for relational databases.

Each object is a member of a **class**, which is a template for creating a certain kind of object. A class contains **instances**, corresponding to real-world entities, and similar to the idea of a tuple in the relational approach. Each object that belongs to the same class has certain properties in common--for example, the same attribute variable names. A class in turn may derive attributes and methods from another class. This type of hierarchical structure results in a property known as **inheritance**. Inheritance reduces the amount of duplicated data by allowing generalizations of data to be made at a higher level and then propagated through the hierarchy. For example, consider a vehicle class which contains subclasses such as car, truck and bus. Information common to all vehicles, such as the fact that they contain tires, engines and seating room can be collectively defined at the vehicle class level. All instances of this class are then known to have these properties. In addition, specialized information such as the number of tires, size of engine, etc., can be defined with each subclass.

Encapsulation is another important OO principle. Encapsulation means that each object contains all of the information regarding its own state (variables) and behavior (procedures or methods). One object can access another object's components only through a well-specified interface; thus, each object acts as a standalone software package that can interact with other objects in some pre-defined way in order to perform a task. This is how OO software supports information hiding, meaning that users of a particular function should not need, or be allowed to know, the details of *how* the function is performed, but only how to invoke it. The advantage of this is that changes in the implementation of a particular function do not imply changes to users of that function.

Composite objects are those objects that are composed of other objects. For example, a car is composed of a body, an engine, tires, etc., each of which is an object in itself. This type of relationship is commonly encountered in the real world, yet difficult to model with a relational database. As described earlier, the relational model uses the approach of foreign keys to maintain this type of dependent information, which results in performance and referential integrity problems. In object-oriented databases, these relationships are maintained through system-maintained direct pointer links; that is, each object contains a direct pointer to any other object upon which the first object is by definition dependent. Moreover, these pointers are maintained by the system, which guarantees: (1) each object identifier is unique, and (2) referential integrity is maintained upon update (e.g. if a referenced object is deleted, all references to that object are automatically updated). These features free the database application programmer from many of the concerns associated with the use of keys in relational databases. In addition, the direct pointer links eliminate the need for costly join operations. An example of an OO model, or schema, of the relational

database example shown in Figure 1 is given below in Figure 2. Figure 3 shows the actual instance relationships of the database, illustrating the direct pointer links which make the use of "join" operations unnecessary.

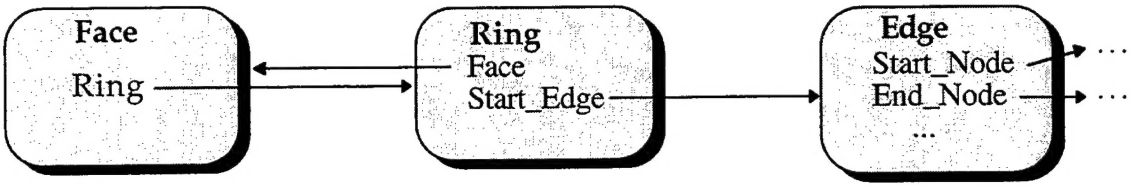

Figure 2. An OO schema (class definition)

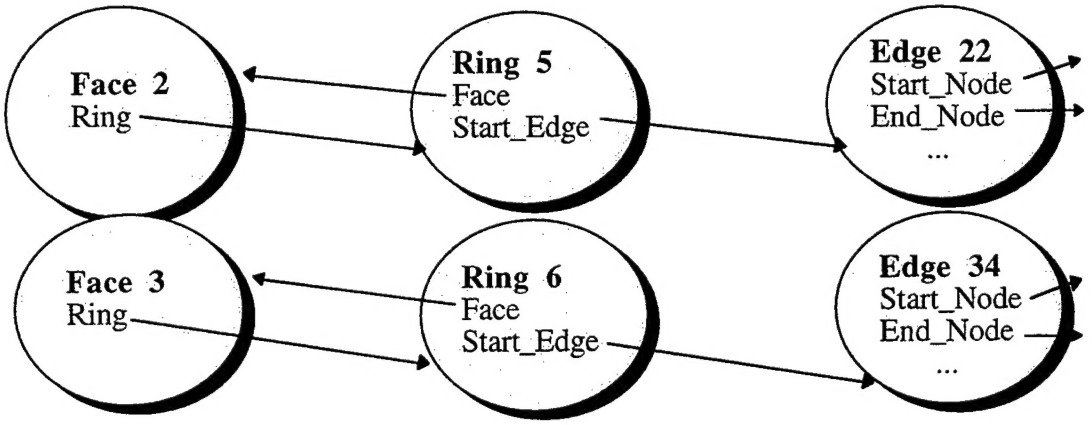

Figure 3. Example OO instances

Object-oriented databases are not as mathematically formalized as relational databases. Because of this, OO technology is sometimes criticized as consisting of a hodgepodge of definitions, with no agreed upon standards. Despite this criticism, there are well-documented foundations for OO work in the literature (e.g. [Kim 90]) and work is progressing rapidly for setting formal standards for OO databases.

Project Background

This project capitalizes on work performed in FY94 for the Defense Modeling and Simulation Organization (DMSO) and DMA [Shaw 94]. The result of this effort was an object-oriented system (ODNC) capable of importing data from a Digital Nautical Chart (DNC) database, a VPF product, transforming the data to an object-oriented structure and exporting the data to an object format. The resulting object database could then be imported by ODNC at about an order of magnitude speedup as compared to the original relational database. (This was with no attempted optimization efforts for performing the relational joins.) Additionally, a viewer and editor with a

point-and-click user interface was incorporated into the system, demonstrating initial capabilities for updating feature attributes.

As of the start of this project, the capabilities of this prototype system were as follows:

- Line and point features from non-browse libraries of DNC01 could be imported, displayed and exported to object format. (This excludes area and text features.)
- Displayed features could be selected (spatial query) via point-and-click interface, and associated attribute information could be displayed.
- Feature attributes could be updated in a limited manner.
- The map view window supported zoom in/zoom out on the display region.

As is documented in the following section, these capabilities have been significantly expanded in the initial phase of this project, and enhancements equal in magnitude to those incorporated thus far are expected for the next phase.

We retained the development system of the DMSO project, which was primarily based on Smalltalk from ParcPlace Systems, Inc. Smalltalk is a pure object-oriented programming language, and ParcPlace Systems' version is coupled with a rich set of development tools which greatly enhances programmer productivity. The use of ParcPlace Systems' Smalltalk also allows for connectivity to at least two leading commercial ODBMS's, Gemstone by Servio, Inc. and ObjectStore by Object Design, Inc., slated as part of the future work for this project.

Initial development efforts were performed on an IBM RS6000 and a Sun SPARC 10 platform. The system was later ported to a Sun SPARC 20 for continued efforts. Power Mac workstations served as front-end systems to the SPARC workstation. Although development work is possible on the Power Mac's themselves, all efforts thus far have been concentrated on the SPARC 20.

III. FY96 Accomplishments

Prototype Development

As stated earlier, the work for FY95 began with the ODNC prototype at its state of development. Major enhancements and additions have been incorporated in terms of expanded functionality, an improved user interface and class hierarchy design.

Incorporation of Area Features

One of the first major accomplishments was the development of an importing and display capability for area features. The ODNC design already included many of the classes needed for the definition of area features. These were easily expanded and utilized accordingly. Similarly, the displaying of area features could be performed almost identically to that of line features. Thus, much of available code were re-used, greatly reducing production time.

For example, by making area features a subclass of line features, much of the needed variables and functionality from the line feature class were inherited. This is one of the major advantages to using an OO approach.

The majority of the work for this effort was therefore centered on the development of the importing function. Methods were created for reading in the area features from the VPF database, which included tracing through the face, ring and edge tables to collect the area boundaries. The capability for exporting area features to object format was already present in the ODNC design and worked well upon testing.

Adding New Features

Another significant milestone was the addition of the functionality for adding new point, line and area feature objects to the map database. Point features may be added by either a point-and-click interface, or by specifying a file from which to read the point's coordinates. Line and area features may be added by specifying a file which contains the coordinates of the line or area boundary and appropriate edge markers.

The addition of new features to the object database was accomplished with very few revisions to the existing program structure. This was due to the OO principle of encapsulation. Encapsulation implies that changes to a particular class or object have little or no impact on other objects and the way in which they function. Therefore, the addition of a new feature which is encapsulated with its data and methods does not produce undesirable side-effects on any existing features. This is not the case with the relational approach, however, as adding a new line or area feature would make it necessary to change, for example, an edge table which is shared by multiple existing line and area features.

To add a new feature, the user first imports features from the desired library and coverage. Then he or she selects the feature type to be added by clicking on one of the feature class types in the list pane, e.g. railrdl. Then the user selects the *new feature* option from the edit menu, and the *point, line* or *area* option from the resulting sub-menu. Selecting *point* results in another sub-menu with *Click to add* and *Read a file* options. If the first option is selected, the user is prompted to point to the location on the map pane at which to add the point feature and click the mouse button. If the second option is selected, the user is prompted to enter a filename from which to read the coordinates. This option is the only one available for line and area features, as it is not practical or accurate to have the user add these features directly through the use of the graphical interface.

Display Functionality

Improvements were also made to the existing map user interface to provide aids for analysis and display purposes. One of these was the addition of feature type "filter" checkboxes. There are four such checkboxes, labeled "Point", "Line", "Area", and "Text". The checkboxes affect the display of corresponding feature classes in the feature listpane; i.e., the feature listpane displays only those types of features for which the associated checkbox is marked. This provides the user with an easy way to "unclutter" the list by displaying only those feature types in which he or she is interested.

A related feature which was incorporated at the same time as the filter checkboxes was the addition of point, line, area and text icons to the feature class names in the feature listpane. The icons provide an immediate visual cue to the user as to which feature type a name belongs.

Two action buttons were also added to the map interface window. These are labeled "Show" and "Hide". The user uses these buttons by first selecting one or more feature classes from the feature listpane, and then clicking on the appropriate button. The "Hide" button erases from the map display all instances of the feature class(es) selected. The "Show" button works in the same manner, except that the semantics are to display the feature instances (assuming they have been previously hidden). These two buttons allow the user flexibility and control in the display of features, rather than being forced to view all features simultaneously.

Feature Editor Enhancements

Major advances were made with respect to the ODNC feature editor, both in terms of its presentation and user interface, and in terms of its functionality. The feature editor is the means through which the user may view and edit information regarding a selected feature. It is accessed by first selecting a feature in the map pane, and then clicking on the "Query" button. To begin, the basic appearance of the editor was changed from a simple window presentation to a multi-page notebook interface. This change was made in order to more effectively present the multitude of feature information to the user in as organized and efficient a manner as possible.

The basic VPF feature editor consists of a five-page notebook. The notebook pages are labeled "Basic", "Attributes", "Location", "Notes", and "Color". The "Basic" page contains information regarding the feature's database and library name, feature class name and description. Any of this information may be modified by editing the appropriate field.

The "Attributes" page contains a table listing the feature's attribute names, values and descriptions. The value and description fields may be edited directly, by first clicking on the appropriate table entry, and then providing the new values in the given input fields. Alternatively, for those attributes which are associated with a standard set of values and descriptions, the user may click with the second mouse button on a look-up table icon, and a pop-up menu of the allowed values and their corresponding descriptions will appear. The user may then select any of the menu items, and the table will be updated with the selected value and description. This feature ensures that only allowable values are entered for updates.

The "Location" page contains a list of the coordinate points associated with a feature. One of the significant additions to the editor was the ability to update a feature's coordinates. Currently, inserting, deleting and modifying of coordinates are supported through the use of action buttons listed below the coordinate table. There is an additional button labeled "Display changes" which will update the feature's display on the map pane in accordance to modifications made to the feature's coordinates. It is anticipated that this feature will be used as one check that the user can perform to identify obvious errors before making the final decision to accept the coordinate modifications. In the future, a "cancel" option will be implemented which will undo any

modifications made; however, at this time, coordinate modifications are made permanent at the time performed.

The "Notes" page contains any feature-related notes as documented in the VPF specification. Notes may be edited by the user through direct interaction with the text pane in which they appear, or may be added in the same manner, if none exist.

The final page is "Color". At this time, the color page is merely a placeholder; however, in the near future the user will be able to specify the desired color for a feature, and to have that feature displayed with the new color.

The feature editor is also the means by which information for a newly added feature is entered. Upon the addition of a feature, the editor is automatically invoked to allow the user to specify all of the necessary information for the type of feature that was added. Noticeable default values (e.g., 0000) are provided for new features in order to alert users to the cases where features have been added, but not properly initialized.

Incorporation of Multiple VPF Products

Perhaps the single most significant achievement to date has been the incorporation of additional VPF products, World Vector Shoreline Plus (WVS$^+$), Vector Smart Map (VMAP) Level 0, and Urban Vector Smart Map (UVMAP), into the existing prototype. This was accomplished primarily through the restructuring of the ODNC class hierarchy into an OVPF hierarchy, along with some factoring of the class hierarchy in order to support the additional databases. Along with changes to the class structure, incremental prototyping changes were made to the map window in order to reflect the accommodation of multiple databases.

The new class hierarchy allows us to take advantage of the OO principle of inheritance, meaning that all attributes and methods which are common to VPF products in general need only be defined and stored once; all VPF databases will then be able to utilize them. Only those properties specific to individual VPF product databases (e.g. DNC01, WVS$^+$, VMAP, UVMAP) need be defined in order to add that database into the model. Because the class hierarchy is the most crucial part of the prototype, an entire section, Section V, is devoted to its description and explanation.

Investigations

In addition to prototype development and design work, investigations into areas that will significantly impact our future work on this project and GGIS as a whole have started. Subjects for which preliminary investigations have been undertaken include:

- the feasibility of using a hybrid (relational-OO) database for GGIS,
- differences in, and possible advantages/disadvantages of various commercial ODBMS architectures

Due to the significance of these issues and their potential long-term impact on GGIS, they will be discussed in detail in Section IV, which is dedicated to the presentation of all key findings to date.

IV. Key Findings

Prototype Findings

Direct updating of VPF products possible

One of the primary advantages in using an OO approach involves the ability to directly update VPF database information. Encapsulation of feature information makes changes to a feature's attribute or location information, as well as the deletion or addition of new features, much easier to perform than would otherwise be possible using traditional approaches. This is a critical finding, since GGIS will need the ability to receive update information from the field, make corresponding changes to the map database and distribute the updated information back to the user in a timely fashion.

Integrated VPF products possible

A major finding from this year's work is the fact that an integrated OO framework for handling multiple VPF products is possible, and has indeed been implemented. As mentioned briefly in a previous section, the capability to import data from the WVS$^+$, VMAP 0 and UVMAP products have been included in the prototype, which previously could handle only DNC data. The effort to implement the ODNC framework took months; the integration of WVS$^+$ then took several days; and integration of VMAP 0 and UVMAP each required only a few hours of programming time. Handling of multiple VPF database product types is essential for the GGIS data warehouse concept. Although some integration issues remain, this initial achievement of uniting four different product databases in a single framework demonstrates the soundness of our approach.

Moreover, the accomplishment of this task at an earlier than expected date, directly demonstrates two of the often cited advantages of using an OO approach—increased programmer productivity and reduced development time.

Performance Benchmarks

A comparison of import times for point and line features from both the A0108280 (Norfolk Approach) and H0108280 (Norfolk Harbor) libraries for the DNC01 database from VPF relational tables versus OO-formatted files was conducted. A complete description of this comparison and the results can be found in Arctur 1995. A summary of the findings is given in this section.

The principal finding of this study was that for the particular libraries involved, a speedup factor of 5.7 to 14.8 was shown for importing from OO-formatted files over importing from relational files. A great deal of this performance improvement was attributed to comparative amounts of file processing time. For example, approximately 3500 files were required to represent data for the combined libraries for the relational approach, whereas only approximately 200 files were required to represent the same information for the OO approach.

Although benchmarks relating to space allocation have not specifically been conducted, this same study noted that the 3500 relational files used about

16 Mbytes of storage space, while the 200 OO files used about 15.5 Mbytes of storage space. Although these results are preliminary, they do indicate that the benefits of OO technology can be realized without incurring penalties with respect to storage needs.

VPF Feature Attribute and Attribute Value Findings

During the course of the work performed to date, several characteristics of feature attributes and values across VPF coverages, libraries and databases have been noted that will make true integration difficult to achieve with existing VPF differences. Primarily, these consist of examples of feature attributes and values which are inconsistently defined for multiple occurrences of a feature class. These findings are also documented in Dallal 1993 and Shaw 1995.

The problem with feature attributes is that in some cases, inconsistent sets of attributes are defined for a given feature class. This means that across databases, and even across libraries and coverages within the same database, a feature class could have either more or fewer attributes than the same feature class defined elsewhere. For example, in the DNC01 database, the *railrd* feature class in the cultural coverage of the H0108280 library has two attributes, **loc** and **vrr**, that are not present in the railrd line feature class in the cultural coverage of the A0108280 library of the same database. While this may be intentional in the data design in order to present less feature detail in smaller-scale libraries, it still clouds the issue of merging coverages from different libraries or databases in response to customer requests.

Another related problem deals with the accepted definitions for certain attribute values. For example, a value of "0" for one attribute could mean "undefined", while a value of "0" for another attribute could mean "unknown", or it could be a valid datum value.

These inconsistencies present a hurdle to accomplishing genuine data integration in the spirit of GGIS. A standard set of attributes should be defined for each feature class, and those attribute values should be given consistent meanings across coverages, libraries and databases. This step will greatly facilitate the progress toward the realization of GGIS.

OO Advantages

In the course of this work, numerous advantages that an OO approach has over traditional programming paradigms have been found. Some of these have been mentioned in the course of this report, including reduced development time and increased programmer productivity. These advantages are directly related to the OO concepts of encapsulation and inheritance.

The principle of encapsulation, meaning that objects are packaged with their data and operations, results in code that is inherently more reusable than traditional code. Because an object can only be accessed by other objects through a specified interface, changes to the internal structure of an object (i.e., *how* an object performs a task) have little or no adverse side-effects on other objects that use that object's services. This means that changes to a particular section of OO code less often result in a cascade of changes that must be made throughout the system. This feature results in a system that is

conducive to an incremental prototyping effort, which allows users to quickly view, refine and make improvements to a working program.

The use of inheritance reduces the amount of duplicated code for a system by allowing data and behavior for related groups of objects to be stored only once; these properties are then *inherited* through the definition of an object hierarchy. Through inheritance, a very small portion of code can be added that immediately has a great deal of functionality. This results in less duplication of code, and therefore, programming effort and time.

Illustra Hybrid Database System

Part of this project included an investigation into, and possible use of an OO-relational hybrid database system in conjunction with the OO prototype. The benefit of using a hybrid database system is that, theoretically, it offers the flexibility and power of an OO approach while maintaining compatibility with standard relational databases.

We investigated Illustra, one of the leading commercial hybrid databases. Inheritance is the principle OO concept which is supported by Illustra. Both "types", which are templates for creating tables (similar to "classes" in OO terminology), and tables themselves can be hierarchically defined to take advantage of inheritance. Both single and multiple inheritance are supported.

Although the hybrid approach appeared promising at first, further investigations into Illustra revealed several serious drawbacks, especially in comparison to a fully OO approach. Some of these are listed below.

- Incomplete support for spatial data needs, for example, there is no support for topological relationships
- Necessity of linking SQL with C functions in order to achieve computational completeness
- Lack of a development environment makes changes difficult, and makes rapid prototyping impossible
- Suffers from traditional relational database problems (e.g. foreign key maintenance, distribution of feature information across multiple tables, etc.)

Our overall evaluation of this product is that, while it seems an excellent alternative for business and traditional database applications, it is not sufficient for complex spatial data needs such as the ones for this project, and especially for GGIS.

ODBMS's

An Object-Oriented Database Management System (ODBMS) can follow one of the three types of client-server architectures: object-server, page-server, and file-server. The types are based on the server since it is the server that maintains and manages the database. The distinction among the three types is based on the unit of data transfer between a server and a client workstation. An object-server architecture understands the concept of an object.

Therefore, each transfer from a server to a client workstation is based on the requested objects. A page-server architecture understands the concept of a page which can range between 256 to 4096 bytes. Each transfer from a server to client workstation occurs in increments of a page. A file-server architecture understands remote file services such as Network File System (NFS). Any access to the database occurs remotely using remote file read and write.

Only the object-server and page-server architectures will be discussed in detail. The two ODBMS's that will be considered provide interfaces to both C++ and Smalltalk object-oriented languages. Since the prototype is implemented using Smalltalk, only those aspects that are pertinent to the Smalltalk interface will be considered.

GEMSTONE

Gemstone includes additional Smalltalk class libraries to provide database management capabilities, i.e., persistency and retrieval. With this extension and a creation of Gemstone Smalltalk Interface (GSI), a client workstation understands the concept of object and therefore is able to apply methods or behavioral actions on retrieved objects. A simple configuration is shown in Figure 4.

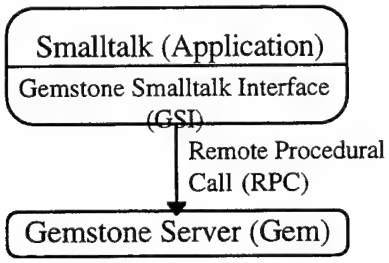

Figure 4. Gemstone configuration diagram

The client interaction is at the Smalltalk(application) level such that any user can request for a copy of an object(s) for analysis or retrieval purposes. The GSI handles the actual processing and interaction between the database server and the client workstations. The GSI caches a copy of the requested object(s) at the user space. This renders a duplication of an object(s); i.e., the same object can be found on the client side as well as on the server side. The changes that are made to the objects on the client side are temporal. However, at the end of a transaction, a close examination and understanding of changes need to be made to synchronize the user space and the database space data information.

The client-server relationship is depicted in Figure 5.

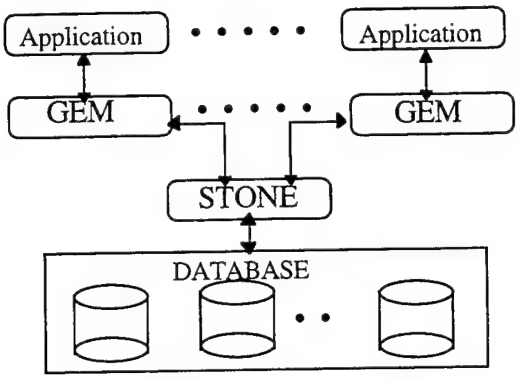

Figure 5. Client-server diagram

The Gem is an object server to the application, but a client of the Stone. Code that is generated by a user is compiled and executed by the Gem. This is a repository of objects retrieved from the actual database--a local repository of requested objects. Any changes and execution can be applied to the objects in the Gem.

An instance of a Gemstone object is called a proxy. Once an object(s) is retrieved from the database and exists in the Gem, the objects are in the local user space. A user maintains a new *symbolDictionary* to maintain name space for each object that was retrieved from the database to allow for later synchronization with the database at the time of commit. Gemstone provides a mechanism to minimize the name space concern. A forwarder can be used that allows execution on a proxy to occur at the Gem instead of in the application.

The Stone is the database monitoring process. It provides services to Gem processes such as allocating object Ids and pages, handling locks, serializing commits, and running garbage collection in the background. The Database is a logical construct of one or more files.

With the two name spaces, a 'dirty read' is possible in Gemstone. Since each application has a copy of needed objects in its own cache, until an actual save request is made, an application makes changes ignorant of other applications' involvement with the objects in question.

Gemstone is an active database in that behavior of objects can be stored and executed in the database, i.e., Smalltalk code executes in the Gem. This can result in a better division of work and resources between the client and server machines. For example, the Gem can be performing authorization and filtering tasks best handled by a centralized process.

Transparency

Gemstone provides four levels of data transparency between the client and the server. This can be seen in the following table:

	Transparent	Non-Transparent
Application	Replicates + Mark Dirty	Lost
GEM	Forwarders	Proxies

Table 1. Data transparency levels

Replicates are duplicate database objects that are stored and manipulated in the application or user space. Any changes that are made will be "marked dirty." Only those objects that are "marked dirty" will be transferred to the database at the time of commit. Data synchronization becomes problematic--any change to an instance variable in one space (user space) is not automatically made in the other space (database space).

Forwarders do not have a copy of the objects. Appropriate messages are sent to the objects in the database. This capability of executing behavior in the database is the key to active objects and active databases. Behavior synchronization becomes problematic--a desired behavior must be defined in the database to execute in the database space.

Object Migration

Object migration deals with synchronizing any references between transient and persistent objects. Any changes on the object based on execution in a transaction must be handled. Object migration is handled by using the *migrateTo:* method. However, potential conflict in terms of other users having access to the same object(s) needs to be anticipated.

Transaction

A transaction is an atomic unit of work that provides a consistent view of persistent data. When a transaction begins, the Gem is given a copy of the current object table and a set of disk pages to use. Any changes are made to this copy and written only to those disk pages. When a transaction commit succeeds, the Stone makes the Gem's copy of the object table the current object table. When the transaction aborts or fails, the Stone gives the Gem an updated copy of the current object table, and any objects created or modified in the aborting session are rejected.

Concurrency

Two types of concurrency control are used for transaction management: optimistic and pessimistic. An optimistic approach assumes conflicts are rare. Recovery is implied to be cheaper than prevention. A pessimistic approach assumes the opposite. Therefore, explicit locks should be used to ensure only one transaction control. Gemstone uses the optimistic approach. This approach provides a couple of advantages: (1) read-only transactions never block; (2) low overhead since locks are not used, and (3) no potential deadlock since a lock failure is notified. However, there are also disadvantages,

namely: (1) commits may fail; and (2) all changes may have to be redone when a commit fails.

Locks

Dirty locks, meaning some user has written and committed a change to an object after another transaction began, are used in Gemstone. The integrity of the object is not reliable. Locks persist across transaction boundaries; therefore a user that began a transaction before a commit occurred needs to abort and refresh the object with the updated objects.

Problem Areas

The major problem that Gemstone has deals with the "two name space" issue, local application space and database space. How to synchronize the two name spaces is a concern. If the two name spaces are not synchronized, all work that may have been done at the user space may need to be replicated after the database space has been updated.

Dirty locks provide a greater problem for database management, especially when one considers that the objective in using a DBMS is to provide data integrity and data quality to many users.

OBJECTSTORE

ObjectStore ODBMS is a non-active database; it functions as a simple repository of data. Figure 6 shows the ObjectStore configuration.

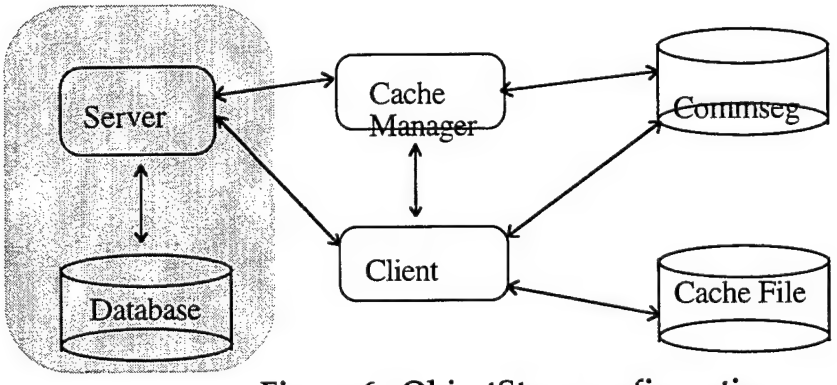

Figure 6. ObjectStore configuration

ObjectStore consist of the server, the client, and the cache manager. The database and the files consist of the cache, communication segment, and the database. The *ObjectStore server* provides I/O services to databases which reside on local and remote disks. Transaction semantics are enforced by providing multi-user concurrency control.

The server uses a two-phase commit protocol in conjunction with other servers to guarantee consistent transaction completion in a distributed database environment. The *ObjectStore client* allocates and de-allocates storage for persistent objects. Communication with the ObjectStore server is managed for fetching and locking pages. Pages are mapped into the virtual address space of the client processes. Recently referenced pages are maintained and coordinated with the cache manager to hold locks as long as possible for minimizing network I/O traffic with the server. Transaction

commits are performed to transmit any modification to the database. The *ObjectStore cache manager* handles asynchronous lock callback requests from the ObjectStore server. This capability provides immediate response to the ObjectStore server which in turn leaves the ObjectStore client library free to make only synchronous requests.

To reduce network traffic with the server and to provide coordination via shared access to the client *commseg* file, a "lazy" lock release mechanism is coordinated with the ObjectStore client. The *ObjectStore database* can be accessed only by the server. However, the client is responsible for database organization and structure. *Cache and commseg files* are used for controlling encached data. The cache file is used as a swap space for persistent data, and as a backup store for in-memory persistent objects. The commseg file tracks the status of encached pages. This file is used by the client to determine whether a page can be accessed and locked. The cache manager uses this file to check and/or give up locks upon the server's request.

Object Migration

A transitive closure on references from persistent objects to transient objects is performed. For each transient object referred to by a persistent object, the following events may occur: (1) migration policy is used to determine how to resolve the cross-space reference; (2) the transient object may be migrated into persistent space, and (3) an object in transient space may be replaced by a different object. The default migration policy specifies a transient instance referred to by a persistent object to be migrated and clustered with the referring object; otherwise, the transient instance is replaced by an instance of the *OSTombstone* class.

Transaction

Any access to, or execution on a persistent object must be performed within a transaction. Any changes made during a transaction are invisible to others until a successful commit has taken place. A transaction can be specified as update or read-only.

Concurrency/Locks

When a client accesses a persistent object, a lock is requested from a server. A single page may have many read locks or one exclusive write lock. A deadlock transaction is detected by a server. The server determines a client to be a "deadlock victim" and aborts the transaction.

Summary

The above descriptions of Gemstone and ObjectStore are based on preliminary research. Each architecture has its advantages and disadvantages, and further study is necessary to determine if one is better suited to the needs of GGIS than the other. Both offer clear advantages over a relational or hybrid object-relational architecture.

V. Prototype Design

The overall schema design consists of thirteen categories of classes, which together provide support for the following: map window/user interface functions, graphics, low-level data support, feature definition, and georelational information. Of these, the most significant in terms of providing for a unified framework are the ones dealing with feature definition and georelational information as shown in Figures 7 and 8.

The general design philosophy for the prototype was to abstract those characteristics common to all VPF products into high level classes, the VPF classes. From these, classes for specifying distinct product features were created as subclasses of the VPF classes.

This principle is illustrated in Figure 7. Figure 7 (A-D) shows the VPFGeoRelational hierarchy, which is concerned with defining the VPF georelational information regarding database, library, coverage and feature organization. As shown in the diagram, specialized subclasses for DNC and WVS+ data exist for each of these except for the library class. Because DNC, VMAP and WVS+ maintain the same library structure, they can share the same attributes and methods in the VPFLibrary class through the use of inheritance.

Figure 7(E) shows the hierarchy for VPFFeature. This class is subclassed into the four types of spatial features supported by VPF: area, line, point and text. Within each of these, both DNC and WVS+ have distinct feature classes. For example, as seen in Figure 7 (E), DNC has bridge area features, asministration line features, building point features, etc., while WVS+ contains a different set of feature classes for each feature type. VMAP and UVMAP involve refinements to the DNC and are subclassed accordingly.

Figure 8 shows the composition hierarchy for a DNC line feature, DNCCoastal. Each instance of a DNCCoastl feature contains attributes concerning coordinates (prims), attributes, notes and its feature id. Additionally, each feature contains a *featureDef* , which is an object of the VPFFeatureDef class that contains information at the feature class level, including the class name, description and feature type, among others. This in turn has a *coverage*, which is an instance of the DNCCoverage class. This class contains coverage-level information such as the coverage name and header information for tables. Each instance of this class also contains a *library*, which is an instance of VPFLibrary. The VPFLibrary class includes library-level information for table headers, tiling, etc. It also holds onto an instance of the VPFDatabase class, which contains information such as component libraries and table header information. This schema allows each feature to know its primitive (location coordinates) information, as well as its containing coverage, library and database information.

In summary, figure 7 supports the inheritance property of OO technology which allows code reuse and compact data structure. Finally, the use of object pointers in figure 9 presents ease in maintainability and manageability as benefits for OO technology.

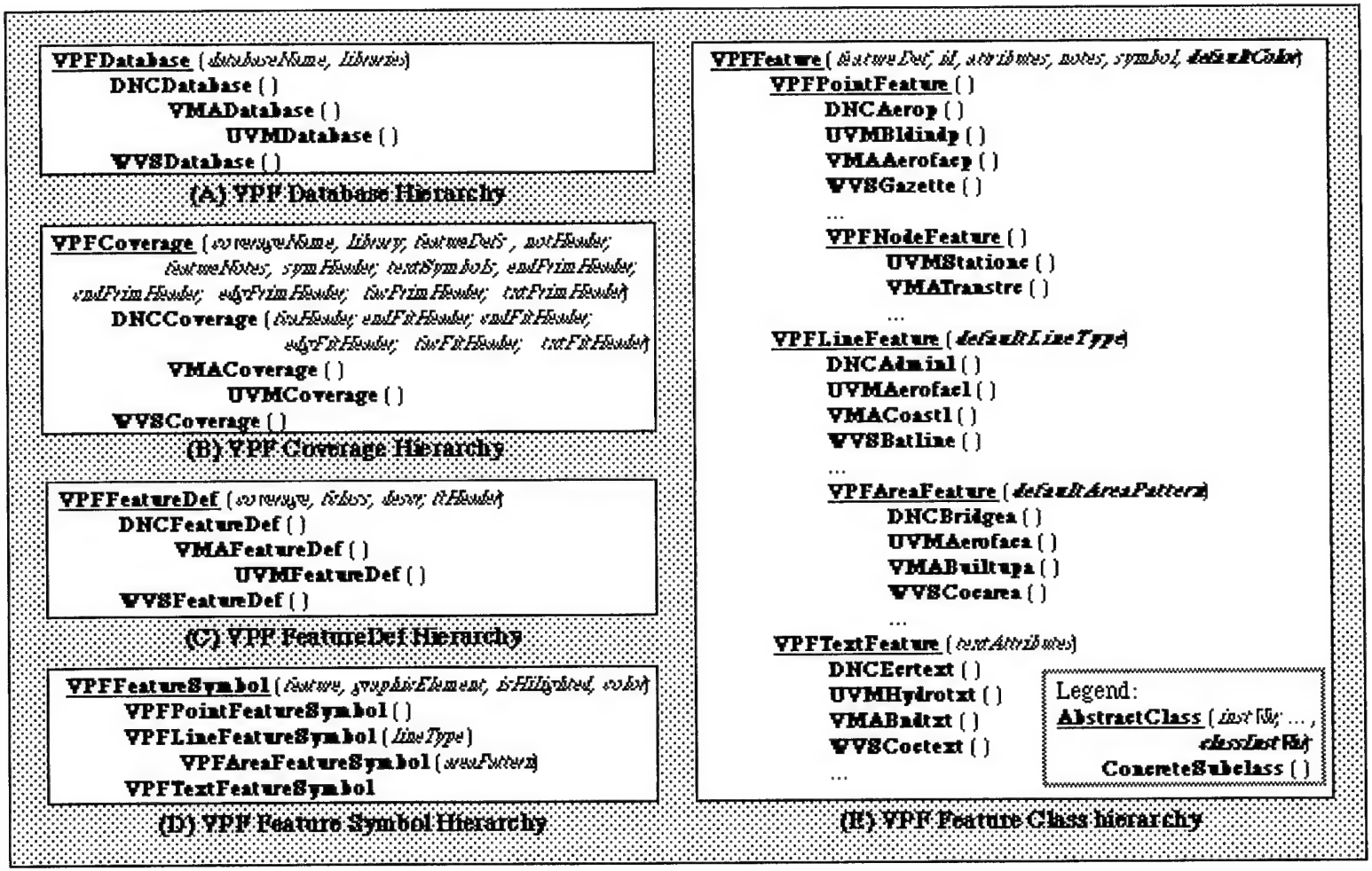

Figure 7 Feature Definition and Georelational Classes

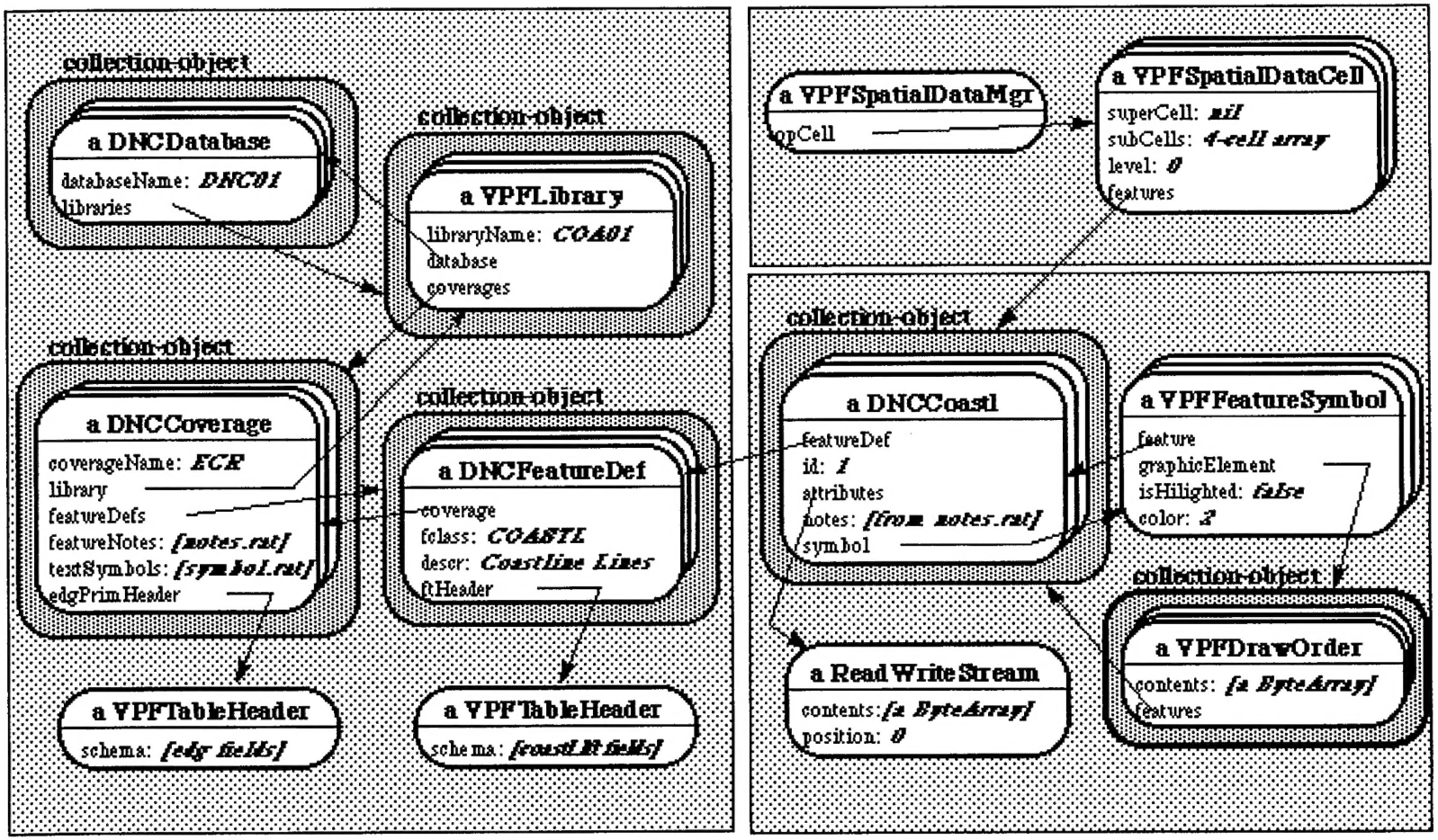

Figure 8 Sample DNCCastl Class

VI. Future Work and Need

As documented in this report, many of the project goals have been achieved. Still, much work remains for the coming months to test and evaluate our prototype. Furthermore, three areas of functionalities need to be implemented and tested:

- provide an option for exporting the objects to a relational format
- implement the rule-based system for maintaining topology across coverages/libraries/databases on update
- provide import, display and export functions for text features

Additionally, the prototype needs to store the OO-formatted VPF data with the commercial ODBMS's Gemstone and ObjectStore using a Smalltalk interface, and, finally, demonstrate an updating capability over a network.

Based upon our involvement with this project, an integrated framework that can accommodate not only multiple VPF product databases, but data from multiple *families* of data (e.g. RPF, TPF) is not only desirable, but is feasible using an OO approach. Such a framework would allow, for example, a vector map to be overlaid on a raster image in such a way that the user would have access to the information content of both, as well as any related TPS information. This implies a level of integration beyond that necessary for simply providing a visual overlay of the different data types. The results of this current project will lay significant groundwork for such an initiative, which would be of tremendous benefit to DMA and to DMA's customers.

VII. Recommendations

These recommendations, listed below, are based on the work presented in this report, including both actual prototyping efforts and academic research.

1. In order to fully realize the advantages of merged product databases, work needs to be performed with respect to standardization of feature attributes and attribute values, as discussed in a previous section. This step will allow users to access coherent, merged feature definitions, rather than a confusing array of attributes and attribute value definitions.

2. Due to the shortcomings of the hybrid database, this approach for GGIS is not recommended.

3. An object-oriented approach is well-suited to the complicated task of integrating multiple databases and data types as is needed to support DMA's GGIS program. The use of OO technology over traditional development methods is highly recommended for the following reasons:

•The OO approach is more capable of modeling complex data types (entities composed of other entities, such as spatial data) than are traditional approaches, such as relational databases, which require that all data be maintained in a flat, non-hierarchical structure.

• OO allows all of the logical components and attributes of a feature to be stored together, rather than spread through multiple data structures (e.g.,

tables). This reduces the number of side effects that could occur, for example, upon update of a feature's attributes or location, or upon the addition of a new feature to the database. The implication of this is that less work has to be performed upon feature update, resulting in fewer complications and potential errors, and improved performance.

•Development and maintenance of OO software is noticeably easier and less time-consuming than that of traditional software. This is primarily due to the application of the OO principles of encapsulation and inheritance. Encapsulation of code and data within an object means that localized changes can readily be made without causing adverse side-effects to other parts of the program. Inheritance results in less code duplication, reducing the amount of development time.

•Import time of object data has been demonstrated to be significantly faster than that of relational data, as documented in Arctur 1995.

While further study us on-going, results thus far have clearly demonstrated the utility of OO in providing information to users and to optimally handle updating and potentially multiple data type integration.

References

Arctur D. K, E. Anwar, J. Alexander, S. Chakravarthy, M. J. Chung, M.A. Cobb, K. B. Shaw, "Comparison and Benchmarks for Import of Vector Product Format (VPF) Geographic Data from Object-Oriented and Relational Database Files", International Conference on Large Spatial Databases, 1995.

Dallal S. L., "Development of a Hierarchical Dual Function Terrain Data Set", SBIR Phase I Final Report, November 1993.

Elmasri R. and S. B. Navathe (1989)*Fundamentals of Database Systems*, Redwood City, California: Benjamin/Cummings,.

Kim W., "Object-Oriented Databases: Definition and Research Directions", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 3., September 1990.

Shaw K., M. J. Chung, M.A. Cobb, and D.K. Arctur, "Development of an Object-Oriented Digital MC&G Database System for Modeling and Simulation Support", Final Project Report to DMA and DMSO, 1994.

Shaw, Kevin, et al., "Digital Mapping, Charting, and Geodesy Analysis Program Quarterly Report, July-September, 1994: Technical Reviews", NRL/MR/7441-95-7568, February 1995.

Smith, David N. (1991)*Concepts of Object-Oriented Programming*, New York: McGraw-Hill.

Taylor, David A. (1990)*Object-Oriented Technology: A Manager's Guide*, Reading, Massachusetts: Addison-Wesley.